PaCE Man

Oscar Araya Garbanzo - 2018002998
Saymon Astúa Madrigal - 2018143188

I Semestre
Año 2020

Lenguajes, compiladores e intérpretes

Área académica de Ingeniería en Computadores

Instituto Tecnológico de Costa Rica

TEC | Tecnológico de Costa Rica

1.1. Descripción de las estructuras de datos desarrolladas.

Position: estructura de datos del código en C que se encarga de contener la coordenada x **xPosition** y la coordenada y **yPosition** de la posición seleccionada por el administrador en su interfaz gráfica en Java. Son utilizadas en la función *checkAvailablePosition* para revisar que la posición colocada por el administrador sea válida según la matriz lógica.

```
struct Position{
    int xPosition;
    int yPosition;
};
```

AnswerToJava: estructura de datos del código en C que se encarga de contener la información referente al mensaje que se quiere enviar al administrador en Java. Se compone de un **messageID** que funciona para identificar al mensaje, un booleano **hasExtraInfo** que indica si cuenta con más información dentro del mensaje, y dicha información **extraInfo**. Este mensaje es enviado al administrador en Java para brindar retroalimentación al querer efectuar un comando, como colocar un fantasma o aumentar la velocidad.

```
struct AnswerToJava{
    char messageID;
    bool hasExtraInfo;
    int extraInfo;
};
```

MessageToServerC: estructura de datos del código en Java que contiene los parámetros con que será ejecutado el servidor en C. Se compone de dos partes, un identificador **messageID** y la posición actual que tiene el administrador seleccionada en la interfaz gráfica **currentPosition**. Estos parámetros son analizados por el código de C, y en función del **messageID** evalúa si es necesario o no el uso de la **currentPosition**. A partir de este mensaje se genera un *AnswerToJava* que es analizado de vuelta. Tiene la siguiente forma:

```
[
String messageID,
String currentPosition
]
```

level1: Representación lógica del mapa de juego. Se almacena en una lista de enteros estática, cuyos elementos representan distintos tipos de posiciones, en donde:

- El 0 es la representación de los muros, los cuales ni el jugador ni los fantasmas pueden atravesar.
- El 1 es la representación de los espacios donde se pueden y deben colocar pac dots al inicio de la ejecución de cada nivel. Este estado es intercambiable por el estado 2, cuando el jugador toma un pac dat en esa posición. Tanto el jugador como los fantasmas pueden circular estas casillas.
- El 2 es la representación de los espacios vacíos. Tanto el jugador como los fantasmas pueden circular por estas casillas también.
- El 3 es la representación de los muros de salida de los fantasmas, casillas que si son atravesables para ellos, pero no para el jugador.
- El 4 es la representación de la casilla de teletransportación izquierda. Al posicionarse sobre esta, el jugador reaparece en la casilla 5. Solo el jugador puede teletransportarse, los fantasmas no.
- El 5 es la representación de la casilla de teletransportación derecha. De igual forma, al posicionarse sobre esta, el jugador se teletransporta a la casilla 4.
- El 6 es la representación de los espacios en donde se debe colocar una pastilla de poder al inicio de la ejecución de cada nivel. Si bien estas son las posiciones definidas por defecto, el administrador es libre de colocarlas en las casillas de tipo 1 y 2, además del 6.

```
// logical representation of the level
private final Integer[][] level1 = {
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0},
        {0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
        {0, 0, 6, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 6, 0, 0},
        {0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
        {0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0},
        {0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
        {0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
        {0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 2, 0, 0, 2, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 2, 0, 0, 2, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 2, 0, 0, 3, 3, 3, 0, 0, 2, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 2, 0, 0, 2, 2, 2, 2, 0, 0, 2, 0, 0, 1, 0, 0, 0, 0, 0, 0},
        {0, 4, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 0, 0, 2, 2, 2, 2, 0, 0, 2, 2, 2, 1, 2, 2, 2, 2, 5, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 1, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 1, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0},
        {0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0},
        {0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0},
        {0, 0, 6, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 6, 0, 0},
        {0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0},
        {0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0},
        {0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0},
        {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
        {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
        {0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
};
```

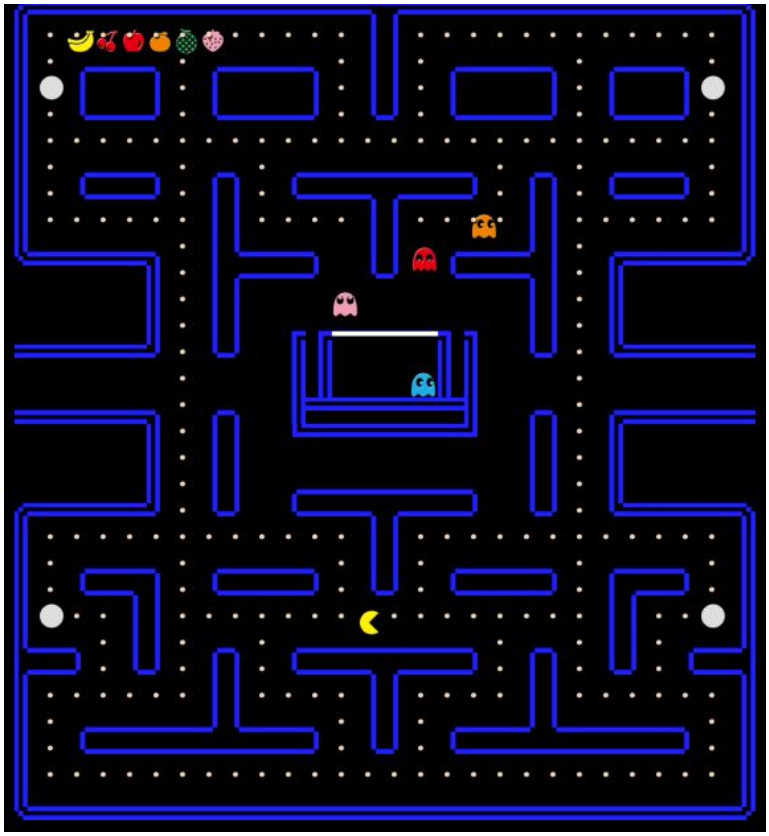
Dato curioso: es posible visualizar el mapa de juego a partir de este extracto de código. Simplemente hay que juntar la vista en un punto central y alejarse un poco de la pantalla o monitor.

Esta estructura es utilizada en diversas funciones dentro de la lógica del juego, como la función del cálculo de colisiones, el método encargado del dibujo de los elementos ajenos al escenario (como pac dots, frutas, pastillas de poder y fantasmas), o las funciones de aumento de puntos. Se puede decir que esta estructura compone la base sobre la cual se realiza todo el análisis lógico dentro de PaCE Man.

Ya que conlleva una cantidad de trabajo considerable, se prefirió enfocar el esfuerzo y tiempo de desarrollo en aspectos como el comportamiento de los fantasmas o la posibilidad de contar con un menú de administrador intuitivo y bien pulido. Se tomó la decisión de únicamente conservar un tipo de mapa, pero si se implementaron los tres niveles, por medio de aumentos en la velocidad y dificultad en general, tal como en el juego original de Pac Man.

Sumado al trabajo a nivel de programación, hay que sumarle el tiempo necesario en diseño. Se creó el primer nivel en base a una plantilla encontrada en internet, pero los otros niveles requerían un tiempo de diseño y planteamiento en GIMP que se prefirió invertir en otros aspectos más relevantes para la experiencia del usuario.

En forma gráfica, este mapa se representa en PaCE Man de la siguiente forma:



SERVER_MATRIX y map: Variante de *level1*. A diferencia de este, aquí solo se contienen las posiciones en donde el jugador y los fantasmas pueden moverse. Los ceros son la representación lógica de las paredes, y los unos son la representación lógica de los espacios válidos para el movimiento. Ambas estructuras, utilizadas tanto en el cliente en Java como en el servidor en C, cumplen las siguientes funciones:

1. En Java, **map** se utiliza para calcular las rutas a seguir por los fantasmas mientras están en modo de “movimiento errático” (no han definido aún una ruta específica por medio del pathfinding A*).
2. En Java, **map** se utiliza para calcular las rutas a seguir por los fantasmas, por medio del algoritmo de pathfinding A*.
3. En C, **SERVER_MATRIX** funciona para validar si la posición seleccionada por el usuario es válida para colocar frutas o pastillas de poder.

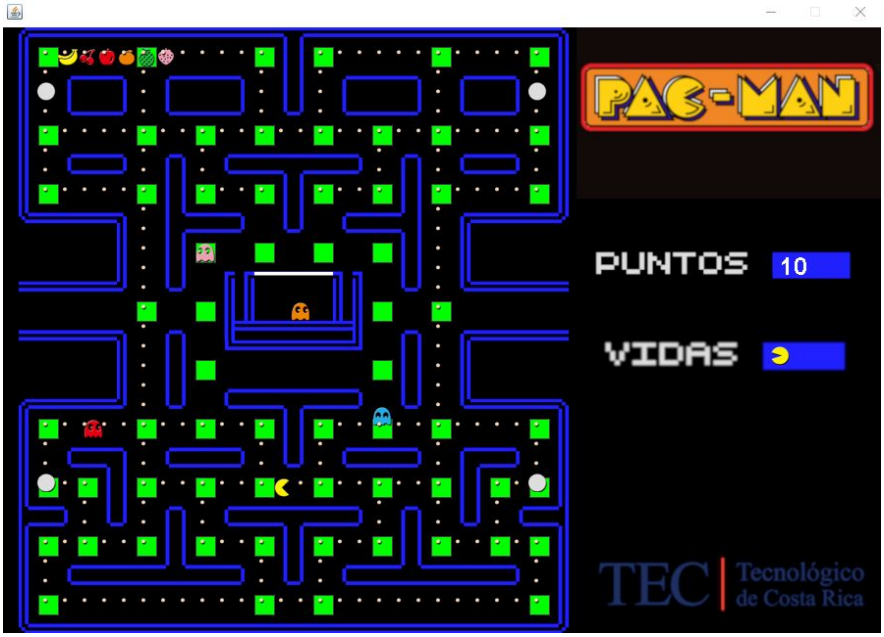
```java
public static final int[][] map = {
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0},
        {0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
        {0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
        {0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
        {0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0},
        {0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
        {0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
        {0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0},
        {0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
        {0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
        {0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0},
        {0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0},
        {0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0},
        {0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0},
        {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
        {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
        {0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
};
```

```c
const int SERVER_MATRIX[31][30] =
        {
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0},
                {0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
                {0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
                {0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
                {0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0},
                {0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
                {0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
                {0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0},
                {0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
                {0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0},
                {0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0},
                {0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0},
                {0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0},
                {0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0},
                {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
                {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
                {0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0},
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
        };
```

1.2. Descripción detallada de los algoritmos desarrollados (como funcionan los fantasmas).

Algoritmo del comportamiento de los fantasmas:
Antes de comenzar con la descripción del algoritmo de los fantasmas, es necesario que se aborden y se definan ciertos conceptos relacionados con el mismo:

Puntos específicos: son los puntos, zonas, o posiciones del mapa en donde se puede realizar un cambio de dirección por parte de cualquiera de los personajes del juego. En la siguiente imagen los mismos se ilustran con un cuadrado de color verde.



Direcciones: los personajes solo tienen cuatro posibles movimientos, arriba, abajo, izquierda, derecha. Una dirección determina hacia qué siguiente posición avanzará el personaje, ya sea los fantasmas (a una velocidad determinada por el administrador) o el jugador (que mantiene una velocidad constante en el tiempo que dure la partida).

Una vez que se tienen claros estos conceptos, se puede proseguir con la explicación del algoritmo. El mismo se utiliza en tres de los cuatro fantasmas (los cuales serán especificados a continuación). Se basa en la búsqueda de una ruta con el fin de encontrar la posición del jugador, y con ello tratar de que este pierda una vida.

Para encontrar una ruta dentro de una matriz o un grafo existen diferentes algoritmos, pero en el caso específico de este proyecto se optó por usar el pathfinding A* . Una de las razones de su escogencia por sobre otras opciones consideradas en un principio

(backtracking o Dijkstra) fue su eficiencia y eficacia para obtener la ruta más óptima entre un punto de inicio y un punto final.

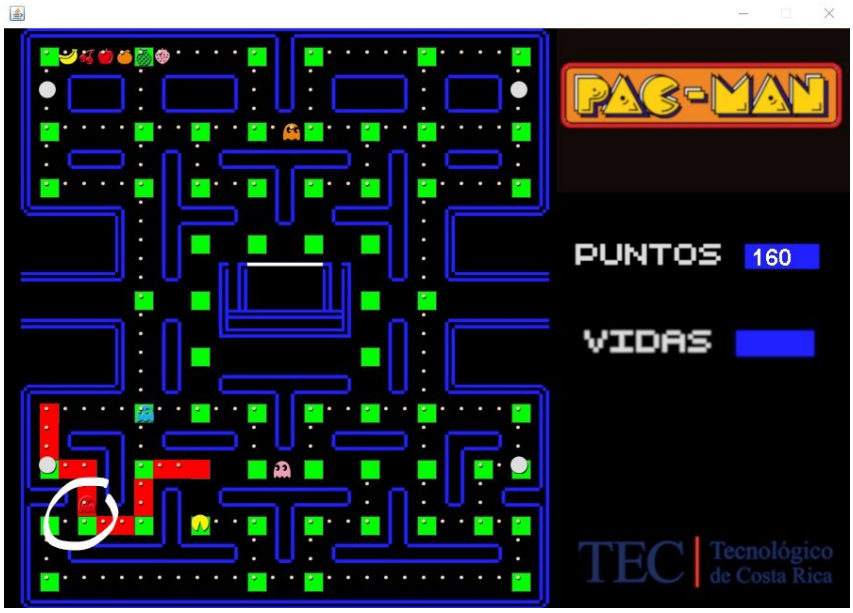
Cada uno de los tres fantasmas que utilizan el pathfinding A* tienen un comportamiento similar, pero cuentan con ligeras variaciones para hacer de la experiencia algo más entretenido e impredecible. Su forma de recorrer el mapa es la siguiente:

Shadow (Blinky): tal como en el juego original de 1980, este fantasma es el más agresivo de todos. Inicialmente busca al jugador por medio del pathfinding A*, usando su posición inicial y la de el jugador. Una vez que cumple con esta primera ruta, el algoritmo se limita a un rango de píxeles para poder ser activado, es decir, que el fantasma inicia su búsqueda del jugador en caso de cumplirse lo siguiente:

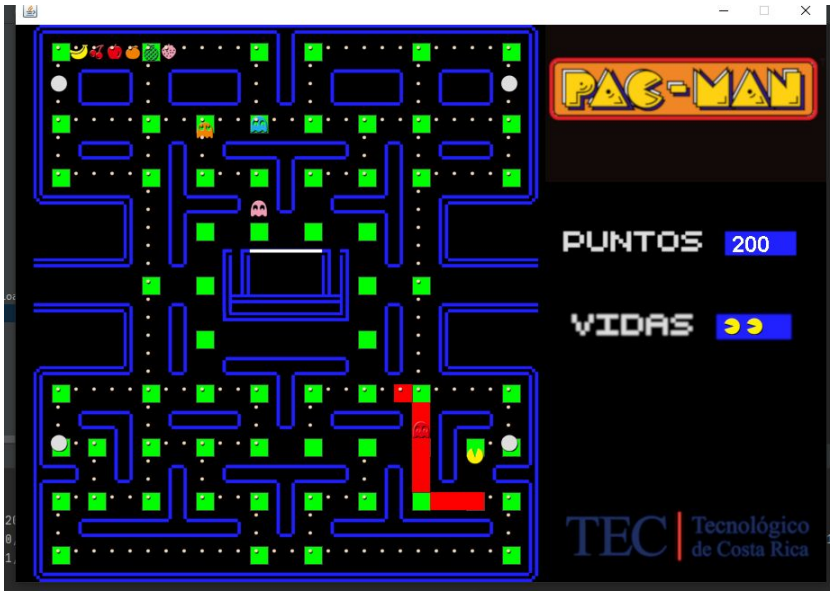
- ➔ Que el jugador haya cambiado su posición en el mapa.
- ➔ Que el fantasma pase nuevamente por un punto específico (puntos de cambio de dirección explicados anteriormente).
- ➔ Que la diferencia de píxeles en el eje entre la posición de el fantasma y el jugador sea de máximo 200, mínimo 30, y que la diferencia de píxeles en el eje y sea de máximo 260 y mínimo 30.

Si esas tres condiciones se cumplen, entonces el fantasma vuelve a calcular una ruta para tratar de encontrar nuevamente al jugador.

En la imágen a continuación se muestra al algoritmo de el fantasma en ejecución. Las posiciones rojas indican la ruta a seguir por el fantasma, ya que detectó que el jugador se encontraba al final de la misma. Se resalta el punto específico en donde el fantasma volverá a calcular su ruta.



Como evidentemente el jugador se mantiene en constante movimiento, Shadow se ve obligado a estar constantemente recalculando su ruta, por lo que es bastante impredecible. En la siguiente imagen se representa otra ruta seleccionada por el fantasma, donde el jugador ha logrado escapar nuevamente.



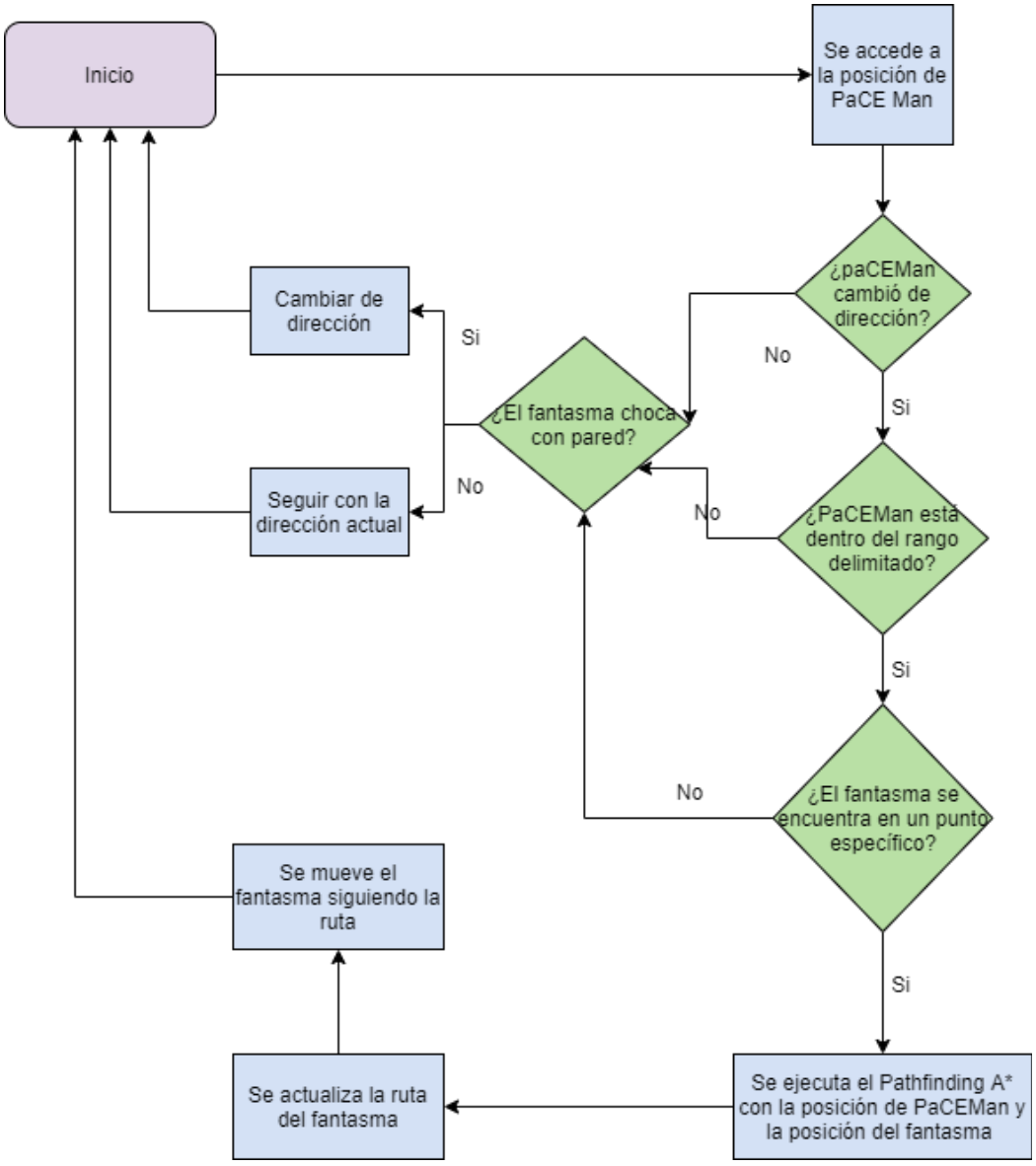
Pero, ¿qué sucede en caso de no cumplirse las tres condiciones mencionadas anteriormente?. Si esto ocurre, Shadow continúa con su ruta actual hasta que se encuentra con una pared, momento tras el cual entra en un estado denominado “movimiento errático”. En este estado, Shadow se limita a recorrer una dirección hasta encontrar una pared, modificar su dirección de forma aleatoria y repetir, en espera de que se cumplan las tres condiciones anterior.

Speedy (Pinky) y Bashful (Inky): su comportamiento es algo similar al de Shadow, pero con variantes en las condiciones para aplicar el algoritmo.

- ➔ Que el jugador haya cambiado su posición en el mapa.
- ➔ Que el fantasma (Speedy o Bashful) pase nuevamente por un punto específico.
- ➔ Que la diferencia de píxeles en el eje x entre la posición de el fantasma y el jugador sea de máximo 140, mínimo 30, y que la diferencia de píxeles en el eje y sea de máximo 140 y mínimo 30.

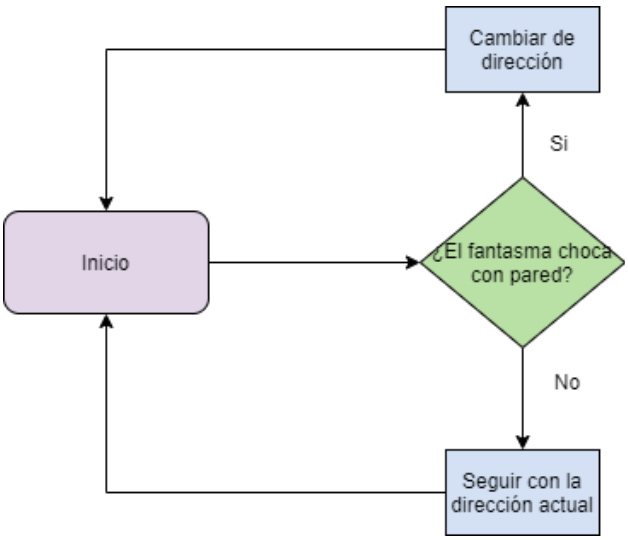
De igual manera si no aplican esas condiciones, ambos fantasmas entran en el estado de “movimiento errático”. La diferencia entre estos dos y Shadow es la distancia en píxeles de activación, haciéndolos un poco menos agresivos, tal como en el juego original de Pac Man. La idea es que estos complementan a Shadow en la tarea de eliminar al jugador, ya que esta diferencia, que puede parecer sutil, cambia enormemente la forma en la que deben ser evadidos.

El comportamiento de estos tres fantasmas que sí utilizan el pathfinding A* puede ser modelado mediante el siguiente diagrama:



Pokey (Clyde): este es el fantasma menos “inteligente” del juego. Al no contar con un algoritmo específico y depender más del azar, su comportamiento resulta en un constante estado de “movimiento errático” que lo hace impredecible y peligroso para el jugador. Al no perseguir de forma directa al usuario, sino recorrer el mapa a su antojo, puede terminar encerrando o sorprendiendo al jugador mientras este huye de los otros fantasmas. Su capacidad de aleatoriedad hace que sea un complemento perfecto para sus compañeros, al aportar el factor sorpresa. Nuevamente, se siguieron los principios básicos del juego clásico, con la diferencia de que este Pokey mantiene su “movimiento errático” durante toda la ejecución del programa.

El comportamiento de este último fantasma puede modelarse en el diagrama que se encuentra a continuación:



Algoritmo de streaming:

Contrario a previos proyectos desarrollados por el equipo de PaCE Man, esta vez se optó por hacer un acercamiento diferente a los sistemas de transmisión o streaming. Por lo general en proyectos de este tipo, se suelen programar clientes con sus funcionalidades limitadas para que solo puedan recibir información pero no interactuar con el juego, dando la impresión de que se trata de un video que reproduce lo realizado en el cliente que si tiene todas las funcionalidades completas.

Esto puede traer inconvenientes varios, como excesiva reutilización de código en el espectador que realmente no necesita, como las funciones encargadas del movimiento de los enemigos o el cálculo de las rutas. Además, el espectador no requiere tanto código en su terminal, ya que el es indiferente en la partida y solo ve lo que sucede. Si bien esta es una implementación totalmente válida, se decidió investigar un poco más y entender realmente cómo es que funcionan los sistemas de streaming de vídeo por internet u otros medios de transmisión de información.

La idea base tras la transmisión de vídeo es que se comunican un grupo de imágenes agrupadas en “paquetes” desde un servidor (en este caso el cliente principal) hasta un receptor (en este caso el cliente espectador). El servidor es un ente “oculto” del receptor, ya que este no participa en las validaciones lógicas o la ejecución de código, y solamente transmite lo que el servidor quiere que sea transmitido.

Basado en este concepto de enviar imágenes por medio de “paquetes” que luego son visualizadas una por una, se implementaron los dos sistemas: el servidor que genera sus imágenes cada cierto tiempo, y el receptor que las reproduce de forma secuencial para generar el video.

Primeramente, en el lado del servidor, se tiene un thread exclusivo al manejo de la trasmisión, que es independiente del resto del programa y no requiere de receptores conectados para funcionar (el juego está listo para transmitir en cualquier momento). Este thread recibe el nombre de Streaming, y su código puede verse a continuación:

```
public class Streaming implements Runnable{
    public Streaming(){
        deletePreviousStream();
    }

    public void deletePreviousStream(){
        File dir = new File( s: "src/GameStream");
        for(File file: dir.listFiles())
            if (!file.isDirectory())
                file.delete();
    }

    @Override
    public void run() {
        while (true){
            try {
                Thread.sleep( l: 50);
                Game.takeScreen();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    };
```

Una vez que el Streaming es ejecutado mediante *run*, constantemente espera 50 milisegundos para transmitir cada una de las imágenes al receptor, las cuales son obtenidas gracias al método *takeScreen*. El código de este método es el siguiente:

```
public static void takeScreen(){
    try {
        BufferedImage image = new Robot().createScreenCapture(new Rectangle( i: frame.getX() + 10,
                                                                              i1: frame.getY() + 40,
                                                                              WIDTH, HEIGHT));
        ImageIO.write(image, s: "png", new File( s: "src/GameStream/"+frames+".png"));
    } catch (AWTException | IOException ex) {
        System.err.println(ex);
    }
    frames++;
}
```

Este método genera capturas de pantalla en la posición y dimensiones de la pantalla de juego, que luego son almacenadas en un directorio local llamado GameStream. Estas capturas de pantalla luego serán accedidas por el receptor, cuyo funcionamiento se compone de un thread exclusivo *StreamReceiver* para esperar 200 milisegundos entre cada imagen reproducida. El código de esta clase es el siguiente:

```
public class StreamReceiver implements Runnable{

    StreamWindow streamWindow;

    public StreamReceiver(StreamWindow framee) { this.streamWindow=framee; }

    @Override
    public void run() {
        while (true){
            try {
                Thread.sleep( l: 200);
                streamWindow.update();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    };
}
```

Finalmente, dentro del receptor de la transmisión, se accede a las imágenes producidas, y se actualizan en pantalla utilizando el método *update* a continuación:

```
public void update() {
    icon = new ImageIcon( s: "src/GameStream/"+ currentFrame +".png");
    streamLabel.setIcon(icon);
    streamLabel.repaint();
    streamWindow.setContentPane(streamLabel);
    currentFrame++;
}
```

De esta forma, se garantiza una transmisión fluida y constante entre el servidor y el receptor, con el único inconveniente de que las imágenes mostradas por el streaming deben tener cierto delay respecto al juego ejecutado. Esto se debe a que la carga computacional de ejecutar ambos programas a la vez hace que las imágenes solicitadas por el receptor sobrepasen a las imágenes generadas por el servidor. Para evitar esto, se implementó un delay de 150 milisegundos entre cada uno de los frames de la transmisión.

Es importante mencionar que en PaCE Man es posible ver una repetición de la partida anterior, al encontrarse almacenada en el paquete de GameStream. Solo basta con ejecutar nuevamente el receptor, y la partida previamente grabada comenzará a reproducirse de forma automática.

1.3. Problemas sin solución.

- Se ha detectado un pequeño bug en el apartado del pathfinding A*. En poquísimas ocasiones, y bajo condiciones que nos fue imposible determinar, el algoritmo ofrece una ruta que hace que los fantasmas no se muevan de su posición actual, haciendo que no avancen hasta que el jugador los elimine.
- La transmisión tiene cierto retraso al reproducir las partidas que se juegan en ese momento. Se debe a la naturaleza misma en como fue programado el sistema de servidor y receptor, al solicitarse imágenes más rápido de lo que se generan. Para evitar fallos en el programa, se optó por generar este delay de 150 milisegundos entre cada frame.
- Es posible forzar al juego a que el jugador atraviese las paredes, al presionar de forma insistente varios botones de dirección a la vez en combinaciones aleatorias. Esto hace que el jugador, en ciertas ocasiones, se comporte extraño a nivel de interfaz gráfica.
- En ocasiones las frutas no se colocan en la posición específica que se seleccionó, pero esto sucede solo con ciertas posiciones, no con todas. En este caso, este problema no se solucionó más que todo por cumplir con el tiempo de la entrega del proyecto, pero si se conoce la manera de solucionarlo.

1.4. Plan de Actividades realizadas por estudiante.

Descripción	**Responsable**	**Tiempo Aproximado**	**Fecha de entrega**
Primera reunión. Lectura de la especificación y análisis de la misma. Recopilar dudas para preguntar al profesor. Asignar tareas.	Saymon y Oscar	1 hora	23 de julio
Investigar Interfaz Gráfica en Java	Saymon	3 horas	24 de julio

Crear los mapas de manera gráfica (Imagen)	Oscar	6 horas	27 de julio
Interfaz gráfica con un mapa jugable	Saymon	10 horas	28 de julio
Implementar algoritmo de los fantasmas	Saymon	8 horas	30 de julio
Crear el servidor en C	Oscar	10 horas	31 de julio
Comunicación por sockets	Oscar	5 horas	2 de Agosto
Crear los 3 niveles	Saymon	4 horas	3 de Agosto
Crear el cliente observador	Oscar	6 horas	4 de Agosto
Crear ejecutables y realizar pruebas	Saymon y Oscar	4 horas	5 de Agosto

1.5. Problemas encontrados.

- Inicialmente se diseñó la imagen para el nivel de un tamaño de 1500 x 1700 píxeles, la cual era demasiado grande para que pudiera mostrarse completa en la mayoría de monitores estándar. La solución fue redimensionar el fondo, que inicialmente estaba basada en una cuadrícula de 50x50 píxeles, por uno basado en una de 20x20 píxeles. Los fantasmas, frutas y el pacman también fueron reescalados a un tamaño de 20x20 píxeles.
- El primer algoritmo para los fantasmas que se implementó, basado en los principios del backtracking, era demasiado complejo de ejecutar y duraba mucho en encontrar una solución, causando congelamientos en la partida. La solución fue la implementación de un algoritmo de tipo pathfinding A*, más eficiente al encontrar las rutas.
- Se tuvo que diseñar una interfaz gráfica para el administrador programada en Java, contrario a la idea inicial de programar en C. Esto fue debido a que las dos

principales herramientas para creación de interfaces gráficas en C, <windows.h> y GTK, no pudieron ser adecuadas a los equipos donde se desarrolló PaCE Man, al presentar diferentes problemas de compatibilidad e importación.

- Al tener que crear ambas interfaces en Java, fue necesario aislar el código en C en un ejecutable aparte que es consultado cuando el usuario envía una señal en el menú del administrador, para cumplir con la implementación requerida de "servidor en C y cliente en Java". Esto dotó al proyecto de una capa de dificultad extra, al interactuar Java-Java, Java-C, y de vuelta Java-Java

1.6. Conclusiones y Recomendaciones del proyecto.

Conclusiones:

- ➔ La implementación el algoritmo pathfinding A*, suple la necesidad de un algoritmo rápido y eficiente. Aparte de esto, al los fantasmas contar con un comportamiento definido y diferenciado entre sí hace del juego una experiencia mucho más disfrutable.
- ➔ Es necesario conocer las limitaciones y facilidades que aporta cada lenguaje y paradigma de programación. Resulta más sencillo el uso de C para ciertas funciones específicas, como la reserva de memoria o la rápida consulta de datos, mientras que Java tiene facilidades debido a su capacidad de heredar cualidades entre clases y su paradigma intuitivo. Por motivos como este es que, en las diversas fuentes bibliográficas, es complicado encontrar información referente a, por ejemplo, interfaces gráficas en C o reserva dinámica de memoria en Java. Cada lenguaje y cada paradigma son producto de una necesidad específica, y deben aprovecharse sus capacidades, evitando tratar con sus carencias.
- ➔ El uso de Java para crear el cliente fue un aspecto ventajoso, ya que este lenguaje, al utilizar el paradigma de orientación de objetos, facilitó mucho la creación de interfaz gráfica y demás aspectos operativos que un juego como PaCE Man requiere. El uso de la herencia y polimorfismo permite reutilizar mucho código en los personajes y entidades, además de que al tratar a cada personaje como un objeto, las colisiones gráficas entre ellos fueron fáciles de calcular y manejar, al comparar de forma sencilla sus coordenadas.
- ➔ La interacción entre distintos lenguajes y paradigmas de programación es viable, común y conveniente. Para este proyecto, Java aportó su flexibilidad en creación de interfaces gráficas y su capacidad, gracias al paradigma de orientación de objetos, de abstraer las características deseadas de un objeto, como lo puede ser una pared, un enemigo o un jugador. Por su parte, C aporta eficiencia, al no ralentizar ni un poco la ejecución de PaCE Man, aún cuando se le hacen consultas múltiples veces por segundo.

- ➔ Resulta crucial, en paradigmas como el de la orientación a objetos, reconocer la capacidad inherente del ser humano a abstraer las cualidades requeridas de su entorno y generar representaciones aproximadas por medio de las clases, y los objetos que de estas se instancian.

Recomendaciones:

- ➔ Entender y saber usar bien los conceptos del paradigma orientado a objetos en un lenguaje como Java, hace que la creación de un juego como PaCE Man se vuelva más sencillo.
- ➔ Agregar sonidos al juego lo vuelve más atractivo e incluso divertido para el usuario. Por cuestiones de rendimiento no se implementaron demasiados sonidos, aunque en un principio se estaba haciendo, pero se recomienda que en futuras versiones el juego se haga más interactivo implementando más sonidos.
- ➔ Se sabe que el algoritmo pathfinding A* de los fantasmas pudo haber sido mucho más complejo y mucho más parecido al comportamiento original visto en Pac Man, pero se decidió por programar esta variante con el fin de cumplir con el plazo de entrega del proyecto.

1.7. Bibliografía consultada en todo el proyecto.

PAC-MAN SOUNDS (2020). Sound effects, ringtones and music from the classic 1980 Namco arcade game Pac-Man:
https://www.classicgaming.cc/classics/pac-man/sounds

Java Game Development Series (2012). Learn how to make your own games in java right here!:
https://www.youtube.com/playlist?list=PLWms45O3n--6KCNAEETGiVTEFvnqA7qCi

Pathfinding, user: paulyv (2016). Simple pathfinder for 2d arrays:
https://github.com/paulyv/pathfinding

Ejemplo de Sockets en Java (2018). Chat básico entre cliente y servidor:
https://parzibyte.me/blog/2018/02/09/sockets-java-chat-cliente-servidor/

Socket entre C y java (2007):
http://chuidiang.org/java/sockets/cpp_java/cpp_java.php

Ejecución de código C desde Java (2012):
https://stackoverflow.com/questions/5604698/java-programming-call-an-exe-from-java-and-passing-parameters

Bitácora:

24 de julio:

- ❏ Se comienza a investigar sobre la creación de interfaces gráficas en Java, con especial énfasis en los tutoriales de Youtube. Saymon. 3 horas.
- ❏ Se diseña el mapa del juego en GIMP. Se debe diseñar ya que debe tener unas dimensiones específicas de 20x20 píxeles por cuadrícula, y los encontrados en internet difieren de estos valores. Oscar. 6 horas.

25 de julio:

- ❏ Se comienza a elaborar la interfaz gráfica del juego. Se crea la ventana principal, se hacen animaciones del personaje y los fantasmas. También se verifican colisiones entre estos. Saymon. 7 horas.
- ❏ Se genera la matriz lógica a partir del diseño del mapa. Se representa cada pared como un 0 y cada espacio vacío como un 1. Oscar. 4 horas.

26 de julio:

- ❏ Se agrega una ventana de menú al juego, se programan las funciones de colisión, se terminan de añadir las imágenes de las frutas y se introducen sonidos al juego. Saymon. 5 horas.
- ❏ Se inicia con el servidor en C. Se investiga sobre el uso de interfaces gráficas en dicho lenguaje de programación, y se van generando las funciones requeridas para el control del juego. Oscar. 7 horas.

27 de julio:

- ❏ Se crea el primer mapa a nivel gráfico, al cual se le agregan las colisiones respectivas. También se le colocan los pac dots y un contador provisional de puntos y vidas. Se añaden las funciones para comer píldoras, que vuelven vulnerables a los fantasmas por unos segundos. Oscar 3 horas. Saymon 7 horas.

29 de julio:

- ❏ Se crea el método que vuelve a iniciar el nivel cuando el jugador pierde una vida. En caso de que no queden vidas, se regresa al menú. Saymon. 2 horas.
- ❏ Se decide implementar una pequeña base de datos para el servidor en C, con el fin de controlar las variables del juego, como la cantidad de puntos y la velocidad actual. Se inicia a investigar sobre el concepto de streaming, con el fin de programar un sistema para ver las partidas sin requerir de ningún análisis lógico de la información (el observador desconoce el estado del juego, solo retransmite lo que sucede). Oscar. 6 horas.

30 de julio:

- ❏ Se comienza a trabajar en el algoritmo de los fantasmas, se implementa un tipo de backtracking que se resulta bastante ineficiente. Además ordena el código y se implementan más herencias con el fin de volverlo más eficiente y reducir su tamaño. Saymon. 5 horas.
- ❏ Se concluye con el servidor en C. Ahora la base de datos también almacena el estado de los fantasmas, representando si los mismos están activos o no en la partida. Se programan las funciones para modificar la base de datos, además de poder hacerle consultas en base al número de línea. Oscar. 5 horas.

31 de julio:

- ❏ Se decide cambiar el comportamiento de los fantasmas por uno más aleatorio, cuando chocan contra algún límite del mapa cambian de dirección. Se utiliza así de manera temporal para seguir probando otras funcionalidades. Saymon. 4 horas.
- ❏ Se comienza a investigar sobre la programación de interfaces gráficas en C. Se evalúan distintas opciones que parecen poco viables, debido a poca flexibilidad e incompatibilidades a la hora de instalar las distintas herramientas, como GTK, o el uso de la librería <windows.h>. Oscar. 6 horas.

1 de agosto:

- ❏ Se implementa un pathfinding A* para la búsqueda de rutas, que se utiliza inicialmente solo con Blinky. Saymon. 6 horas.
- ❏ Se toma la decisión de que el sistema de streaming funcionará por medio del paso de imágenes. Se programa un prototipo funcional que toma las imágenes que recibe y las muestra cada 200 milisegundos, dando la impresión de ser un vídeo en movimiento. Oscar. 5 horas.

2 de agosto:

- ❏ El algoritmo pathfinding A* se integra a los demás fantasmas, pero este se utiliza de forma distinta. Ya todos los fantasmas funcionan correctamente. Saymon. 7 horas.
- ❏ Tras previa autorización del profesor del curso, se inicia la creación de la interfaz gráfica del administrador en Java y no en C, y debido a esto debe replantearse la arquitectura de todo el sistema. Ahora el código en C pasa a ser un ejecutable al que se le hacen consultas que, en función de la base de datos, genera respuestas que serán comunicadas de vuelta al administrador. Oscar. 3 horas.

3 de Agosto:

- ❏ Se detallan funcionalidades del juego y se implementan los socket del cliente en la parte de Java. Saymon. 4 horas.
- ❏ Se termina de programar el menú de administrador. Se programan los distintos botones que comunican con el código de C, y se reciben las respuestas respectivas. Se conecta también este menú con la base de datos, para que muestre en la interfaz los puntos, la cantidad de vidas y la velocidad actual. Oscar. 5 horas.

4 de agosto:

- ❏ Se detallan funcionalidades orientadas a la lectura de información que el servidor envía y que el cliente en Java debe interpretar cuando le llegue. Saymon. 2 horas.
- ❏ Se programan los sockets del servidor, y se preparan los mensajes que serán enviados al cliente. Se diseñan las funciones que analizarán los mensajes que se reciben del cliente. Oscar. 4.5 horas.

5 de agosto:

- ❏ Se documentan las partes de código que faltaban a nivel interno y se trabaja en la documentación externa. Saymon. 3 horas.
- ❏ Se completa el sistema de streaming. Ahora el cliente genera imágenes que el espectador puede consultar y mostrar en pantalla, dando la impresión de tratarse de una transmisión en vivo. Se realiza la documentación interna del código. Oscar. 4 horas.

6 de agosto:

- ❏ Se termina la documentación externa. Se generan los ejecutables. Saymon y Oscar. 6 horas.